Master Software Engineering — Thesis

Master Software Engineering

UvA Universiteit van Amsterdam

Risk Reducing Test Case Generation for Model-based Testing

A feasibility study

J.E. Hendriks

August 31, 2010

Supervisors: Machiel vd Bijl (Axini B.V.) and Paul Griffioen (UvA)

Acknowledgements

I would like to thank Axini B.V. for giving me the opportunity to do this research at their company. In particular I would like to thank Machiel vd Bijl for his extensive knowledge on the subject and his support in writing this thesis. I am grateful to Mark Timmer, who as a co-founder and author on risk reduction based testing, explained this theory. And last but not least I am grateful to Paul Griffioen as my supervisor at the University of Amsterdam.

Abstract

Using automatic testing methods, like model-based testing, it is possible to generate, execute and confirm many tests. The number of tests can be infinite due to the state explosion caused by loops and unranged data structures. It is impossible to test an infinite number of tests. However this can be tackled using a coverage method and measure. The problem with the use of coverage is that every action in the system being tested has equal importance. In practice this usually is not the case and the way around therefore is to define and select the important parts of a system, making a better choice of a finite amount of tests from an infinite state space.

Stoelinga and Timmer published a paper ([ST09]) on defining and measuring risk and risk coverage. The theory can be used as a new way of generating test cases. The theory is risk reduction based testing. It provides a way of defining the importance of all parts of the system. In theory this gives a possibility to test important parts of the system and determine whether or not it is worthwhile to continue testing.

A feasibility study was performed in order to validate the theory. The theory was implemented and compared with random testing and a transition coverage method. The experiments showed promising results for the quality of the tests generated by the risk reduction based testing method. The results show risk reduction based testing is able to steer the test cases to risky locations and thus reduce a higher amount of risk. The coverage of risk reduction based testing however is worse than the coverage for the random and transition coverage methods. This indicates a problem with the algorithm which may need solving. We also investigated the performance of the algorithm. The results show that the algorithm is too complex and time-intensive for practical use and needs further research.

Contents

CHAPTER 1

Introduction

1.1 Problems on testing

The first program people write is usually a "hello world" program. A simple program that is not very likely to contain bugs. After a while the programs get bigger and the first bugs or defects arise. Solving bugs requires debugging which takes a lot of time. We thus want to detect bugs as soon as possible. Detecting bugs is done by testing the software [McC04].

Tests can be added when problems show, when the code has just been written or as test-first programming[McC04]. People tend to write tests when there is a problem in their code. The tests are then used to make debugging easier. If a part of the software works there is no need to examine it when looking for the cause of the defect. Better practise is to use tests to find defects.

A test consist of generating, executing and validating a test case. This is a time consuming job, especially when the program is large and contains loops. The state space of a program explodes when loops or data structures are added. Some loops can even cause an infinite large state space because they never end. Writing enough tests for all the different cases takes a lot of time.

Keeping the tests up to date requires a lot of time. Testing a program and keeping the tests up to date would be easier when the three steps of testing are automated. Simple frameworks are capable of automating the executing and validating steps of a test. There are advanced testing techniques that are capable of automating the three steps of testing. One of these methods is model based testing.

1.2 Model based testing

Model based testing uses a model of the program to generate tests and situations [DJK+99, Tre08]. These tests are executed on the real program and the response is checked with the model. This process makes it possible to perform more tests in the same period of time then when testing by hand.

Even though model based tests can generate, execute and validate more tests in the same time as non automated methods, it is still impossible to test everything due to the state space explosion.

There are different ways of selecting which tests would be the best choice to test. Random testing is a way of easily testing and getting reasonable results [DN84]. Statistically random tests can find all the defects. However because of the randomness there is a chance no defects will be found at all and the amount of duplicate tests can be rather high[DN84].

1.3 Adding coverage

To make sure we are able to find defects and reduce the amount of duplicate tests we can add a coverage measure to the test case generator. Coverage is a way of defining what has been tested, and what can still be tested. Using the knowledge of the coverage measure we can generate test cases that would add something to the set of test cases we have already tested.

Using a coverage measure helps us reduce the amount of duplicate test cases but using a coverage measure does not help us to indicate what is important to test or how much risk there would be involved trying to test a specific choice. Timmer and Stoelinga developed a new theory on measuring and defining risk [ST09], it is called risk reduction based testing.

1.4 Risk reduction based testing

Risk reduction based testing is a way of defining risk for every possible test case using the chance of detecting a defect and the chance the system is actually defective. The defined risks can be used to generate test cases that reduce the highest amount of risk and therefore have the highest chance of finding a defect.

Risk reduction based testing has not been used in practice. In theory risk reduction based testing should generate test cases that find more important defects than test cases generated using other methods.

1.5 Research Question

The risk reduction based method is still only a theory and we will therefore do a feasibility study on the quality of the test cases generated by the risk reduction algorithm. We will perform this feasibility study by implementing the risk reduction based testing method in a model based testing framework and compare it against a random selecting method and a transition coverage selecting method.

To compare the different selecting methods we had to define a measure of quality. The comparison will be performed by measuring different values based on the output of the test selecting methods. These results are compared using four quality attributes. These quality attributes give us a way of objectively comparing the different methods. The research questions involved with our feasibility study are separated in one main research question (RQ) and three sub research questions (SQ1-SQ3). These questions are:

RQ: Are the test cases generated with the risk reduction based method of a higher quality than the test cases generated by a random or transition coverage generator when the tests are constrained by time or number of tests.

SQ1: Is there a good way to determine the initial risk values?

- Using a uniform start value
- Using the knowledge of an expert
- Using a random distribution

SQ2: How is the risk reduction test case generation method influenced?

SQ3: How much overhead is there when using risk coverage?

1.6 Hypothesis

To answer the main research question we have first formulated the hypothesis for the sub questions. The hypothesis are:

SQ1 We expect that the initial risk can be best determined by an expert. He does need to have knowledge of the system and its previous defect areas. When taking the input of an expert the risk values should not be set to extreme values, as the expert can make a mistake and taking extreme values would extend the mistake over the tests. When no expert is available a uniform distribution would be the best input because it does not prefer any area, but would have an advantage over non risk reduction based testing due to the adaptive behaviour in between the tests.

SQ2 The error and fail values make it possible to direct the traces in the test set to certain parts of the system under test. When the error and fail rates are of extreme values the test set will contain a slim set of traces and actions. Furthermore when directing the tests to existing errors it will find more errors, simply because it is directed to the errors.

SQ3 Overhead consists of three parts: calculating the risk, choose the highest risk reduction trace and recalculating the risk values after a test has been run. The paper gives a clue on the overhead as it states that the algorithm for calculating the risk has a complexity of $O(n^3 + pm\log(m) + p^3)$ where n is the amount of states, m the amount of possible traces and p the amount of tests cases already executed. This is only a cubic complexity but the amount of states can grow to amounts far over a thousand states when performing a normal test. Thus indicates the overhead for calculating the risks is large.

Taking into account the hypothesis for the sub questions we can now formulate the hypothesis for the main research question:

RQ: When we compare random/transition coverage with the risk reduction based method we will expect to see the traces of the risk reduction test to be of a higher quality than the random or transition coverage methods. The randomly generated tests will have partial or complete duplicates in the traces whereas the coverage test will only cover a certain part of the system. The risk reduction test however will test the parts of the system that are classified as important and when under constraints, the risk reduction test will execute the best tests. If there are defects in the important part of the system these defects will be detected with less traces than with a random or transition coverage test.

1.7 Structure of the thesis

The introduction gives some basic information about the problem and the solution. A detailed description of model based testing can be found in chapter 2. It explains model based testing according to Tretmans ([Tre08]). The chapter also explains why there is a need for risk reduction based testing.

Chapter 3 gives a detailed explanation on how the risk reduction algorithm works and how it tends to solve the main problem outlined in chapter 2.

In chapter 4 we will explain how we will conduct the experiments. We will give a description on the environment in which the experiments will take place. We will also explain how the different methods will be compared using the quality attributes.

Chapter 5 contains the results of the experiments described in chapter 4. The results are processed according to the quality attributes and the results are discussed for their validity.

In chapter 6 we will give an answer to the research questions from chapter 1 using the results from chapter 5. The answer to the research questions will also be used to reflect on the usability of risk reduction based testing. This reflection explains if the risk reduction based testing method really is an answer to the problem described in chapter 2. Chapter 6 also contains possible future work of the risk reduction based testing method that need investigating

CHAPTER 2

Model Based Testing

Automatic tests are the way of reducing the work required for generating, executing and checking tests. Simple automatic test frameworks can execute and check tests automatically. More advanced testing frameworks can also generate tests. Our research area is the automatic testing framework based on models: Model based testing.

2.1 Model based testing

Model based testing is a specification based testing method. A model represents the implemented system. The model is used to generate tests that can be compared with the observed behaviour of the implemented system [Tre08, DJK+99]. This implemented system is called the Implementation Under Test (IUT) or System Under Test (SUT).

An important advantage of model based testing is the ability to reduce testing time and implementation costs. A model based testing framework is capable of testing more than conventional testing methods. The model gives information about how the IUT works. The model makes it possible to test the IUT. To use a model based testing framework a specification of the IUT is needed. The specification is converted into a model and used by the framework.

The test cases that have been generated by the model based testing framework are executed on the model and the IUT. The output of the IUT is compared against the model and a test is said to have failed if this output differs from the expected output.

Failing tests can be caused by a defect in the IUT or in the model. Therefore detected defects need to be checked with the specification to know if they are defects in the model or in the IUT. There is also the possibility defects are not detected, this is usually because the model has the same defect as the IUT. The model and IUT then have the same defective behaviour.

2.1.1 Models

An important part of model based testing is the model. The model is a representation of the IUT but does not implement the actual software. It only represents the different possible actions the IUT can perform. A way of representing the model is with a labelled transition system (LTS), specifically the input/output labelled transition system (IOLTS). An IOLTS is a 5-tuple (S, I, O, R, s_0) where S is a set of states, I is the countable set of input lables, O is the countable set of output labels, $R \subseteq S \times (I \cup O) \times R$ is the transition relation and $s_0 \in S$ is the initial state [VDBRT04, Tre08, DJK+99].

An instance of relation R is called a transition. A transition contains two states and a label. This label represents an action the system or model can do. In an IOLTS the label also contains information on the action type. The type of an action denotes if the action is received or sent, in other words it tells if the action is an input or output action.

Figure 2.1 contains an example of a simple input/output labelled transition system. The states are represented by S0, S1 and S2. The labels are written as an action followed by a question mark for input actions or an exclamation mark for output actions. There is a special

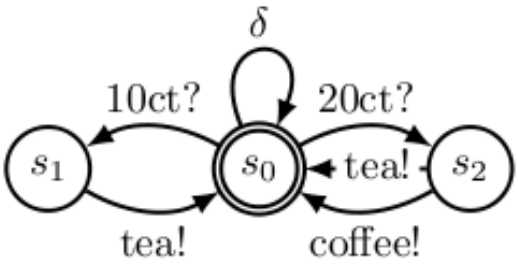

Figure 2.1: An input/output labelled transition system representing a simple tea and coffee vending machine

action that has no type. It is the delta action. A delta action is used to identify that the system does not produce an output in a specific state.

When we are testing we look at the labelled transition system as the state space of a system. It contains all the states a system can be in and the actions to reach these states. A concatenation of actions is called a trace and is used to represent single test case. After every action the system will be in a state. Thus executing a trace results in the system being in a certain state. Because a trace leads to a state, a trace can be used to represent a state, however a state can be reached by different traces. This implies a state does not represent a single trace.

In figure 2.1 the possible actions are throwing in money (10ct? and 20ct?) and receiving something to drink (tea! or coffee!). A simple trace could be 10ct?-tea!. This trace would result in the system being in state S0. This is the same state as the trace started. These loops make the countable transitions and states result in an infinite amount of traces as some traces can always be extended by adding a loop at the end of the trace.

2.2 Testing Constraints

Model based testing makes it possible to test every possible state and action in the IUT producing an output. However testing every possible combination of states and actions can cost an infinite amount of time due to loops in the model[DM90]. This means it is not possible to test every possible option and thus creates the need for testing constraints: what to test and what not to test.

A way to constrain the tests is by systematically running every possible test until a predefined amount of tests has been executed. Instead of executing the tests systematically it is also possible to execute randomly selected test cases. Selecting the test cases at random has the advantage of covering the whole state space. It has the disadvantage that there is a chance nothing important is tested or no defects are found because of the random selected test case [DN84].

2.2.1 Coverage

Just testing as mentioned in the previous paragraph helps choosing a test and executing it. It does not help in knowing how much has been tested or how much should be tested. This can be accomplished by measuring the coverage. Coverage is a way of measuring the amount of tests that are already executed and the total amount of tests available[ZHM97].

Because there is an infinite amount of tests possible the coverage measures constrain the possible test cases. The way the coverage constraines is defined by the way it measures. If a coverage measures the lines of code that have been reached using a test the constraint is the total count of lines of code in the system. When measuring coverage on a labelled transition system the constraint is usually the depth of the test. The depth is the amount of actions performed in a test and is thus the length of a trace. Another constraint on labelled transition systems can be the amount of states reached.

When a coverage measure is used we can test until a certain percentage of the coverage has been reached. The best case would be 100% but depending on the coverage this would not

actually tell if the test result was useful. If for example a trace coverage has a depth limit of 100 transitions it would be impossible to detect defects that need at least 100 transitions to occur. An example of a defect that could occur after more than 100 transitions would be an overflow defect in a calculator or an overflow of the stack space when using large amounts of data.

2.3 Problem

The main problem of testing is not knowing if it would be worthwhile to generate and execute another test. Coverage tells us that we have tested up to 100% of a predefined part of the IUT. But we have no idea if we defined the correct part of the system and if there are not any defects left in the rest of the IUT.

To solve this problem we need a way of defining what is important, and how important it actually is. A new theory from the hands of Brinksma, Briones, and Stoelinga [BBS06] and Stoelinga and Timmer [ST09] gives us a way of defining what is important and how important it is. Their theory lets us assign a risk to every transition. They provide an algorithm that calculates the highest risk values and thus how much risk could be reduced.

The risk reduction values can be used to generate an optimal set of tests that has the highest reduced risk. Before running a new test it can tell how much risk could be reduced with the test. Thus the risk reducing method is capable of solving the problems of coverage in model based testing.

CHAPTER 3

Risk Reduction Based Testing

Risk reduction based testing is a way of defining the importance of every part of an IUT and calculating the test cases with the highest risk and most likely to contain defects. This method is based on the theory of Stoelinga and Timmer in [ST09].

3.1 Defining Importance

The main part of the risk reduction based testing method is the way it defines how important every part of an IUT is. A part in the IUT is defined as a trace or a part of a trace. The importance is eventually defined as a risk. The risk is based on the chance the trace will result in a defect when executing the trace.

The chances for risk are based on three different chances. The first chance is the chance of receiving a certain output action when the testing framework has performed a trace. For example the LTS from figure 2.1 can return tea or coffee after you put in 20ct. By default this is will be defined as a 50/50 chance. However it could be more likely that the chance for coffee is 80% and the chance of receiving a cup of tea is only 20% because the machine will always return tea after putting in 10ct. This chance is defined as P_{out}.

The second chance is the chance we would actually detect a defect after executing a test case. The chance of detecting a defect is less than 100%. The defect will not be detected in the case the model contains the same defect as the IUT or if the defect returns a value that is within a valid output range even though it is not valid. This second chance is defined as p_{fail}.

The last chance that has to be defined is the chance something is actually defective. This chance is defined as p_{err} and is the most important chance the algorithm uses. This chance is used to define the importance of a trace or transition.

Using the same value for every type of chance would result in a uniform system where everything is just as important. The values proposed by Stoelinga and Timmer are 0.6 for p_{fail}, 0.01 for p_{err} and 1 / number of output actions for P_{out}. To define what is important and what is less important to test we use these chances and give everything we think is important a higher chance of failing or detecting.

For an example we will use a simple calculator. It can only add and subtract one and begins at a total count of zero. There are a few things that could go wrong and we want to make sure they are tested. Possible defects are negative numbers and overflow defects. To test the implementation we need at least one trace that results in a negative number and one trace that results in the maximum size of an int or long plus one. To do this we give the trace 0!;-1?;-1! a higher p_{err} to make them more important than the rest of the IUT. We will double the chance it is defective.

The trace $0!; (+1?; result!;)^{max\_int+1}; max\_int+1!$ would also get a p_{err} value of 0.02. In this example we will also reduce the value of p_{fail} to 0.05 because the chance of detecting an overflow defect is smaller. This is because there is a chance both the model and the IUT use the same data type and both result in a defective overflow.

Condition	p_{fail}	p_{err}
Default Input action	0.6	0.01
Default Output action	0.6	0.01
+1?	0.6	0.01
0!;-1?	0.6	0.02
0!;(+1?;result!;)$^{max_int+1}$max_int+1!	0.05	0.02

Table 3.1: Example risks

3.2 Calculating Risk

When all the chances have been defined in a uniform manner or adapted to the situation, the risk reducing method can calculate the risk for every transition in the system. This is accomplished with the algorithm as defined in algorithm 1 which is slightly adapted from the algorithm in [ST09]. The original algorithm returns a test set containing k test cases, our implementation only returns one test case. Our implementation is included in appendix A. We had to make our implementation only generate single test cases because we are not able to force an output action and have to generate test cases while executing the test. We will explain the algorithm broadly. For a more detailed explanation we refer to the paper of Stoelinga and Timmer, [ST09].

Algorithm 1: Computing an optimal test case of depth d

Input: A WFS$^+$ W=$\langle A_{\text{spec}}, w, p_{\text{err}}, p_{\text{fail}}, p_{\text{out}}\rangle$, a depth d and a set of executed test cases T
Output: An A_{spec} with risks defined for every transition in traces$_{A_{\text{spec}}}$

Initialization

For all traces $\sigma \in \text{traces}_{A_{\text{spec}}}$ of length d, set $M_T(\sigma, 0) = 0$

Construction

while $n \geq 0$ **do**

For all traces $\sigma \in \text{traces}_{A_{\text{spec}}}$ of length n, compute

$M_T(\sigma, d-n) = \max(\text{doInput}, \text{observe})$

where

$$\text{doInput} = \max_{\sigma a? \in \text{traces}_{A_{\text{spec}}}}^{a? \in L_I} M_T(\sigma a?, d-n-1)$$

$$\text{observe} = r(\sigma, \text{obs}(\sigma, T)) + \sum_{\sigma b! \in \text{traces}_{A_{\text{spec}}}}^{b! \in L_O} M_T(\sigma b!, d-n-1)$$

$$r(\sigma, m) = c(\sigma, m) - c(\sigma, m+1)$$

$$c(\sigma, m) = w(\sigma) \sum_{i=0}^{m} \binom{m}{i} p_{\text{reach}}(\sigma)(1-p_{\text{reach}}(\sigma))^{m-i} \text{PEP}_W(\sigma, i)$$

$$\text{PEP}_W(\sigma, n) = \frac{(1-p_{\text{fail}}(\sigma))^n p_{\text{err}}(\sigma)}{(1-p_{\text{fail}}(\sigma))^n p_{\text{err}}(\sigma) + 1 - p_{\text{err}}(\sigma)}$$

and set $M_T'(\sigma, d-n) =$

$$\begin{cases} \{\perp\} & \text{, if } M_T(\sigma, d-n) = \text{observe} \\ \max_{\sigma a? \in \text{traces}_{A_{\text{spec}}}}^{a? \in L_I} M_T(\sigma a?, d-n-1) & \text{, otherwise} \end{cases}$$

$n = n - 1$

end

The input for the algorithm is a weighted fault system (WFS). A WFS is a tuple W=$\langle A_{\text{spec}}, w, p_{\text{err}}, p_{\text{fail}}, p_{\text{out}}\rangle$. A_{spec} is an IOLTS. w is a weight function assigning a fault weight to every trace of A_{spec}. In practice this is a depth severity value. p_{fail} and p_{err} are the chances as defined in section 3.1.

The first step of the algorithm is to calculate the risk reduction value for the longest traces in A_{spec}. To calculate a trace the algorithm requires the risk reduction value of the child traces. A child trace is a trace that starts with the same trace as the one we are calculating but adds one extra action. For example a?b!c? and a?b!d! are child traces of a?b!. The algorithm will set the risks of unknown child traces to zero.

After calculating the longest traces, with a risk of zero for their child traces, the algorithm calculates the second longest traces with the previous calculated child traces. This is repeated until the smallest trace is calculated.

The actual risk reduction values (M_T in the algorithm) are calculated by taking the maximum value of selecting an input action or observing an output. Adding an input or output action to the current trace results in a child trace. These child traces have already been calculated and just need looking up.

Every action would add extra risk for its child traces as they represent the risk of those traces. The action to reach the child trace would only add risk if it is an output action. Tests can only detect a defect after an output action. Input actions only put the model and IUT in a specific state, and thus only add the risk of its child traces.

Output actions can not be forced but they do add risk. This is where we need the P_{out} value. This value is used to sum the risks of output actions with their corresponding ratio. The sum of output actions can now be compared to the individual input actions to get the action with the highest risk.

Calculating the risk for an output action requires the calculation of the posterior error probability (PEP_W) and the chance we can actually execute the trace (p_{reach}).

The PEP_W is calculated by using the chance a trace is a defect (p_{err}) and the chance we will actually detect the defect (p_{fail}). The PEP_W also takes into account how many times we have already executed this action in previous test cases.

To make sure the length of a trace is taken into account there is also a depth severity value that is used to define how valuable it is to run deeper tests over shallow tests. It also makes sure all the risks count up to a finite amount as the depth severity value is reduced with a factor equal or larger than the number of child traces.

3.3 Reducing Risk

The output of the risk reduction algorithm is a tree or list with all the traces and their maximum reducible risk. Using this tree or list we can make choices based on what action would result in the best results. As we can only select input actions we must hope the IUT will return an output at points that have a high risk for an output action.

Because it would be bad behaviour for a program to never return an output we expect every trace to eventually result in an output action and thus resulting in an amount of reduced risk. When there is no output action the risk will not be reduced and the risk reduction based test will run the same trace again. Only when there are risks of the same value or when there is an output action will the system select another input action at random. The uniform distribution of chances results in a high amount of actions with the same risk reduction value.

The amount of reduced risk is calculated by taking all the individual transitions in a trace and summing the risk they reduced when they were executed. This results in an amount of reduced risk.

To generate a set of test cases that produce the highest possible risk we select the traces with the highest risk. This already takes into account the chance an output is received. But it could theoretically result in a test set that never receives an expected output. However it would still direct the tests to important parts of the system and if an output action is not received it will continue with the action resulting in the next best result.

The risk reducing method can be used as a coverage measure. If the amount of risk reduced reaches a certain threshold we would stop testing. It is also possible to calculate the total amount of reducible risk and use that total to get a percentage of risk we reduced. We will however not take the coverage into account with our research.

3.4 Complexity

The original algorithm as described by Timmer and Stoelinga in their paper [ST09] has a complexity of $\text{O}(n^3 + pm\log(m) + p^3)$ where n is the amount of states, m the amount of possible traces and p the amount of tests cases already executed.

In a normal testing environment the amount of states and traces can grow rather fast when extending the length of the traces. The amount of states grows when the depth of the tests becomes higher. If there are more states there are also many more possible traces because in theory every trace can have a next transition going to every possible state. In a simple case containing 10 states and traces with a length of 5 transitions the maximum amount of traces would be 10^5. Combining the possible large growth of the traces and states with the cubic complexity we have an indication the algorithm is expensive.

Depending on the implementation this complexity can become a real problem if it is not optimised. An optimisation to add is caching. The algorithm now recalculates every child and it recalculates all the chances up to the current transition. The chances can easily be cached when the list of traces is built and the risk reduction values of the children can be stored in a large list of calculated traces. Other optimisations can be implemented in language specific data types like a tree structure.

CHAPTER 4

Experiment

4.1 Why have an experiment

The risk reduction theory is only proposed in a paper and has never been tried in practice. We will therefore conduct experiments to measure how good the risk reduction based testing method is compared to other test case generation methods. This comparison is done against the random based and coverage based testing methods, both generally accepted as good ways to generate test cases.

4.2 How to experiment

The main part of the experiment is to measure the quality of different test generating methods. To do this we defined four quality attributes that all measure a different quality of a method.

All the tests are performed in a framework. This framework is a program that generates, executes and validates test cases for a model and an IUT. The test cases are executed on the model and the IUT and compared based on possible input actions and received output actions. The test is said to have failed if the output generated by the IUT is invalid or if the input produced by the framework is unknown to the IUT.

The way the framework will generate test cases differs with every method. The random test generator will select one of the actions at random. The transition coverage method is implemented as a local transition coverage method. In every state it reaches it will select an action from the actions that have not yet been performed. When all the actions have been performed it will also select an action at random.

The risk reducing method needs to prepare its data before being able to make a decision. This preparation is done using only the model. After the preparations the risk reducing method will select actions based on the amount of risk an action reduces. Because the preparations cost time and the other methods do not require any preparation time we will also measure the time required to prepare the data for the risk reduction based method.

4.3 Measuring results

The quality of the methods is measured against quality attributes. These quality attributes (QA) provide a way of objectively looking at a testing method resulting in a valid way of comparing different methods. To make sure the different methods can be compared with these quality attributes it is only allowed to compare the methods if they are tested on the same model and IUT containing the same defects.

4.3.1 QA1: Amount of defects found

We are comparing test case generation methods. The test cases are meant to find defects and thus a good way to compare the methods is to count the amount of defects they find. Quality attribute 1 defines the amount of defects found with a test set. We need to count the number of test cases (in our case the number of traces) and the amount of defects these test cases find.

Because a defect can be detected more than once in a test set we need to define unique defects. A defect is an output action from the IUT that is not available in the model. When testing we do not know it the defect will always occur or only with a specific trace leading to the output action. Therefore we define a unique defect as the trace and malfunctioning action resulting in the defect.

In case we have more traces leading to the same defect we will assign them as not being the same defect. This is due to the way we test on an actual system. We do not know if two different traces producing a defect are actually the same defect and because all the methods have the same model and IUT they also have the same possible traces to a defect.

4.3.2 QA2: Point of interest coverage

When testing an IUT there are parts of the system that are more important than others. For example when testing the vending machine in figure 2.1 it is not very likely the machine will output coffee or tea when doing nothing (a delta action in state S0), but testing the machine to return tea after throwing in 10 cents is important.

Quality attribute 2 is meant to measure the number of traces that performed actions on important points of the IUT. Important points of the system are defined as a specific state of the system or a concatenation of one or more action or transitions. If the point of interest is a state this quality attribute counts the number of times the state is reached in a test set. If the point of interest is defined as a concatenation of transitions or actions this quality attribute counts the number of subtraces in a test set that are equal to the point of interest.

4.3.3 QA3: Amount of risk reduced

The risk reduction theory is not only a method of selecting the actions with the highest risk, it is also a way of defining the quality of a test set. The risk reduction theory can be used to measure the amount of risk that is reduced using any method. It is also possible to calculate the total amount of risk and give a percentage of reduced risk.

We will measure the reduced risk of the different test case generation methods using the risk reduction theory as a measure. For risk reduction itself this is easy but for the random and transition coverage methods we will need to add a way to measure risk reduction. Because risks change after every test case we will use the risk reducing method for random and coverage but change the way the actions are chosen according to the method. This will result in a reduced risk value that can be compared between the different test case generating methods.

4.3.4 QA4: Test set Coverage

To determine how well a test set covered the possible actions we need to define a simple coverage measure. There are three things we can measure in model based testing. The amount of traces, the amount of transitions and the amount of states that have been executed. Because we know the maximum amount of states, transitions and traces a test set can reach we also have an indication of the amount of duplicates.

Each of the three coverage measures tell us something else about the system. Trace coverage gives an idea how likely the chance will be the next trace is a duplicate of earlier ones. Coverage on transitions tells how many possible actions have been tested. Comparing on transition coverage gives information on how good a test generating method can choose transitions that have not been executed yet. The coverage on states gives an indication if all the states of a system have been reached. A low coverage on states could be an indication that some states are unreachable.

4.4 Experiment model and IUT

For our experiment we will use the ETE system as testable implementation. To make sure we are able to get consistent results and are able to add defects to the experiment we use a software simulator of the ETE system and have a model that represents the simulator. The ETE system is a communication protocol used in a real system. Because of the size of this system the preparations for the risk reduced tests take a lot of time. We had to test with a depth of eight actions because it was impossible to perform enough experiments with a larger depth due to the preparation time. However we expect this not to be a problem because the use of a simulator as IUT makes it possible to define defects in the system at any location.

We will define three kinds of defects. The first defect will be a specific transition which always fails. The second defect will be a trace that results in a defect. And the third defect will be a non deterministic defect that occurs sometimes. These defects are used to test the ability of the methods to find different types of defects and to take a final measurement in finding all the defects.

4.5 Experiment variation

We will run two different sets of experiments.The first is a set of experiments to measure timings. These timings are used for section 5.1 and were all run 20 times to get an average. The second set of experiments was used to get information for our quality attributes. These tests were also run 20 times, each containing 50 traces to get an average.

Because we wanted to test the steering quality of the risk reducing method we varied the input chances. The normal risk reduction test had a uniform distribution of chances. For the aimed versions of the risk reduction algorithm we used 6 points of interest where we found defects during the first set of experiments. The aiming is performed by entering higher chances for the points of interest. The slightly aimed test used values of 0.01001 for p_{err} for the points of interest instead of 0.01000 for the non interesting points. The heavily aimed variant used a p_{err} of 0.02 for the points of interest and the extremely aimed variant used 0.05 as p_{err} for the points of interest.

4.6 Gathering data

To collect enough and valid data we need to run every test multiple times. The framework will keep track of all variations in the experiments. Starting at the beginning of the experiment the seed for the random number generator is stored to make the experiments reproducible. The framework will then generate a graph containing all the possible states and transitions, the so called state space. The state space will keep information about the failed traces and the number of times a transition is tested.

For the experiments the framework stores information about the executed traces and measure the time required for different parts of the experiment. With every executed trace the reduced risk is stored and the result of the trace. If a trace failed the trace is also stored in a separate data structure containing the trace and the action that resulted in a failing test. This is required because there is no state available after the test failed.

To keep track of the number of times a transition is performed we have a data structure containing all the transitions. A transition is stored as a label as well as the two states it connects.

All the data collected during the tests is stored in separate files, making it possible to process them automatically after running the experiments.

4.7 Validation

Before running these experiments we need to validate the experiments and the possible results. They should be usable for to answer the research questions and the experiments should be

reproducible.

Our main research is to examine the quality of risk reduction based tests in respect of existing methods. This is done with all the quality attributes. They all give a measure of quality that can be used to rank different methods. The other reason for our experiments is to see if risk reduction is able to steer a test case in a specific direction. This is accomplished with quality attribute two.

Even without steering, risk reduction should have a higher quality of tests. This can be validated using the first and last quality attribute. The first tells us the percentage of defects found and the last quality attribute tells us how many transitions were spilled. Random is expected to spill most, as it selects its transitions at random without knowledge of previously executed transitions. The transition coverage method should have the lowest spill because it tries to select transitions that have not been tested before.

CHAPTER 5

Results

5.1 Measuring risk reduction preparation time

The first set of experiments was to measure the time it took to calculate the risk reductions values. These timings were measured using a ruby benchmark tool. This made it easy to run multiple tests and get detailed information on every test and averages. With every test we also measured the amount of traces to calculate the time per trace.

After the first few measurements we noticed an above linear growth in all the values. This growth forced us to limit the experiment to a state space depth of eight and a risk reduction calculation depth of nine. The results in table 5.1 give an idea of the growth in time required for performing the calculations. For a preparation depth of nine we need to calculate the risk reduction value for 2637302 traces which takes approximate 7193 seconds of cpu time and 7730 seconds in real time. This is about 2 hours of work averaging at 2.931 milliseconds per trace.

depth	# traces	user+system time(s)	real time (s)	time/trace(s)
1	1	0.000000	0.000303	0.000303
2	2	0.000000	0.000488	0.000244
3	9	0.000000	0.000932	0.000104
4	52	0.010000	0.010016	0.000193
5	371	0.140000	0.137446	0.000370
6	3076	2.200000	2.207341	0.000718
7	28029	31.430000	31.567511	0.001126
8	268582	429.000000	438.234957	0.001632
9	2637302	7193.00000	7729.997166	0.002931

Table 5.1: Time required to calculate the risk reductions values. *We measured the amount of traces for every depth as a quantity measure. For every depth we measured the time the cpu needed (user+system time) and the time needed including everything else (real time). The timings indicate an above linear growth which can also be seen in the last column time per trace.*

5.1.1 Discussion on timings

After we implemented the algorithm we have the idea the calculations are mostly dependent on the amount of traces and transitions. We need to calculate every transition in every trace to know the amount of risk the trace can reduce. This means the calculations and the time it needs to calculate depend on the amount of transitions we calculate. This is equal to the depth times the number of traces as every trace contains as much transitions as the depth.

After running the experiment we divided the amount of traces by the timings and as we can see the first two measurements seem to be out of order. This can be due to the small amount

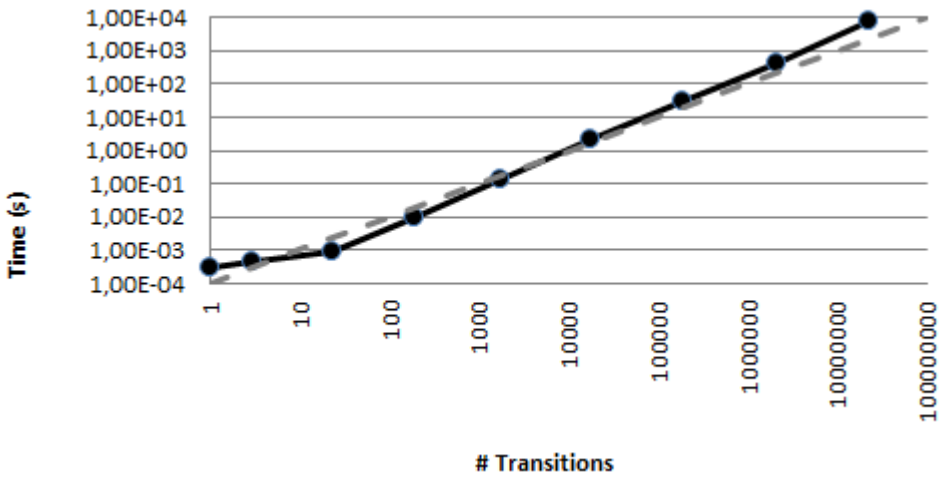

Figure 5.1: The amount of time required to calculate the traces of a given depth in our model. *The dashed line represents a linear growth and the black line represents the growth of the measured values (dots). Because traces have a length of the depth they reach and every depth requires a transition we plotted the total amount of transitions (#traces × depth) against the time.*

of time required to perform the first two depth calculations. The other results do add up nicely with a logarithmic scale.

The next thing we did was plot the results in a graph, see figure 5.1. On the x-axis we plot the amount of transitions (traces × depth) that where calculated. This gave a better indication of the amount of calculations than the amount of traces because the length of the traces increased with an increasing depth. On the y-axis we plot the time needed to calculate the risk reduction values for the traces. We left in the first two measurements and the graph also shows they are out of order. In the graph we can also see there is an above linear growth for the amount of time required to calculate the transitions.

For every tenfold of the amount of transitions there is a fourteenfold in the amount of time required to calculate the risk reduction values. This would result in the following formula to get the amount of time when inputting the amount of transitions: $3.0\text{e-}05 \times 14^{log_{10}(\#transitions)}$.

The formula indicates that preparing for the use of risk reduction based testing can be a problem. It would take only 1.8e08 transitions to get a preparation time of more than a day. In table 5.1 we see that a depth of 9 already has 2.6e06 traces which is approximately 2.6e07 transitions. In table 5.1 we can also see that the amount of traces grows with approximately a factor 10 every depth which means that taking a depth of 10 actions would already result in a preparation time of more than a day and a depth of 11 would take more than a week of preparation time. Because a normal test set contains test cases with a depth which is much higher than 10 using the risk reducing method with a normal testing depth would take a lot of time with the current algorithm. For example if we extend the ten-folding on the amount of transitions we would need more than 1.0e97 years to prepare for a trace depth of 100 transitions.

5.2 Quality Attributes

After calculating the risk reduction values we collected data on the quality of the tests. To do this we added three defects to the ETE system. These defects result in at least six points where a defect can occur. We will use these defect points as points of interest for QA2.

5.2.1 QA1: Amount of defects found

All the methods were capable of finding the three types of defects. They did have different traces to these defects. We measure the amount of unique traces that lead to a defect. To know how

well a method is able to find defects we take the amount of unique defects and divide them by the amount of unique traces we executed. These results are shown in table 5.2. We also added the sigma for our experiment for the amount of unique defects divided by the amount of unique traces.

Method	# unique defects	# unique traces	# defects/trace	sigma
Random	16.6	42.68	0.389	0.059
Transition coverage	13.8	39.79	0.347	0.043
Risk Reduction	17.9	38.53	0.465	0.067
RR slightly aimed	18.1	38.84	0.466	0.067
RR heavily aimed	18.0	38.58	0.467	0.073
RR extremely aimed	18.7	38.52	0.488	0.065

Table 5.2: QA1: Defect analysis after 50 traces. *The amount of defects per trace is calculated by taking the unique defects divided by the amount of unique traces. We then calculated the sigma for the amount of defects per unique trace in a same manner. We can now see that the amount of defects that risk reduction detects is significantly higher than random or transition coverage.*

The results in table 5.2 show a higher rate on finding defects for risk reduction than random and transition coverage. The difference is more than one sigma, and thus significant. The results show that transition coverage is the worst way of finding as many defects as possible. This may however vary when using a different coverage method or model where the defects are located elsewhere. The transition coverage method systematically walks through all the transitions of the IUT, therefore its defect find rate will depend on the location of the defects.

Our explanation for the higher amount of defects that is found using risk reduction is due to the fact risk reduction will always try to generate test cases that have output actions in them. Random and coverage do not use this knowledge and will therefore have less output actions in their test cases and thus a smaller chance of finding defects.

Because the difference is significant we may conclude that the amount of defects found with risk reduction is higher.

5.2.2 QA2: Point of interest coverage

After running the first experiments we knew where to find the defects we put into the system. It was then possible to aim the risk reduction test to these parts of the system and measure how well this aiming works. We defined six points of interest, and measured how many traces each of these points of interest reached. The results are in table 5.3. Every point of interest (POI) is defined as a defect we added to the IUT.

We chose defects as interesting points but we could just as easily have chosen non defect points of interest. Using defects gave us the possibility to see if specific defects where detected and how the risk reducing method handles steering to defects without.

Method	POI 1	POI 2	POI 3	POI 4	POI 5	POI 6	no POI
Random	11	6	3	0	0	0	25
Coverage	12	1	2	0	0	0	25
Risk Reduction	11	3	0	1	0	0	19
RR slightly aimed	12	7	1	0	0	0	19
RR heavy aimed	11	8	1	0	0	1	18
RR extremely aimed	10	10	1	4	1	1	16

Table 5.3: QA2: Amount of traces reaching predefined points of interest(POI). *The more we aim at points of interest the more of them we actually reach with our experiment. This means aiming works if the amount of aim is enough.*

We had expected all the aimed risk reducing methods to find the points of interest. This is not the case and after inspecting the risks produced by the algorithm we identified the cause

why risk reduction methods that were not aimed heavily enough did not find all the points of interest.

If a point of interest is given a higher p_{err} value this propagates through the tree as long as the risk reduction value is high enough to be chosen as the action with the highest possible reducible risk. Because risks accumulate there are cases where the trace to a point of interest has a lower reducable risk than other traces. This lower risk is due to the number of output actions in the child traces and the depth severity value (w). The points of interest with a to low propagating risk will not get selected by the risk reducing method and therefore do not show up in the list of reached points of interest. Using high p_{err} values makes the points of interest propagate much further giving them a higher chance of being selected, hence the results of the extremely aimed risk reducing method.

5.2.3 QA3: Amount of risk reduced

During the experiments we kept track of the risk reduction measure using the framework for risk reduction. This made the test run of random and transition coverage take more time, but it gave a nice way of comparing the risk reducing method on its own strength. The results of risk reducing method are significantly better than transition coverage. This can be seen in table 5.4, where random and transition coverage have a much lower percentage (less than 90%) than all the risk reducing methods have. The amount of risk for every method seems very low but if we compare it to the trace coverage in table 5.7 in section 5.2.4 we see the risk is about 20% of the coverage. 20% of the trace coverage is not bad because a trace can be executed multiple times, each time adding a new amount of reduced risk.

Method	Amount of risk reduced	% of RR	Sigma
Random	2.580e-03	81.8%	3.79e-04
Transition coverage	2.785e-03	88.3%	2.80e-04
Risk Reduction	3.155e-03	100.0%	4.41e-04
RR slightly aimed	3.244e-03	102.8%	4.47e-04
RR heavily aimed	3.216e-03	101.9%	4.87e-04
RR extremely aimed	3.315e-03	105.1%	4.62e-04

Table 5.4: QA3: Amount of risk reduced calculated with the risk reducing method. *Risk reduction based methods all have a high amount of risk they reduce because they select test cases with a high reducible risk.*

Inspecting the amounts of reduced risk in table 5.4 we note that the risk reducing method is able to defeat non risk reducing methods. The differences are significantl between risk reduction and non risk reduction based methods. This is about the same result as we noted in table 5.2 for QA1. The local minimum for the values when using a heavily aimed risk reducing method is also identical. However random has a worse score when compared using QA3 than when compared using QA1. We expect this to be caused by the higher coverage that transition coverage has (See QA4).

5.2.4 QA4: Test set Coverage

For the coverage we measured the amount of unique traces, transitions and states for every experiment. To get an idea of how good a coverage is we also calculated the maximum possible coverage with our experiment. The total amount of states in the model is 1277, the total amount of traces is 268582 and the total amount of transitions is 6197. We used these numbers to calculate the coverage of our experiments. The coverages of the experiment can be found in tables 5.5, 5.6 and 5.7. Each table has a count on the unique states, transitions or traces depending on which coverage we measure. They also have the coverage percentages and a sigma for the unique count.

Risk reduction based methods have a significantly lower transition coverage than the random tests or the coverage method (see table 5.6). This is expected because certain transitions have a high risk and will be executed more often reducing the transition coverage. Because there are

Method	# unique states	Coverage	Sigma
Random	27.68	2.17%	1.887
Transition Coverage	29.63	2.32%	2.006
Risk Reduction	25.32	1.98%	2.829
RR slightly aimed	26.11	2.04%	2.233
RR heavily aimed	26.00	2.04%	2.427
RR extremely aimed	24.21	1.90%	2.440

Table 5.5: State coverage. *The results for state coverage are identical for risk reduction and random. Transition coverage is however significantly better than risk reduction.*

Method	# unique transitions	Coverage	Sigma
Random	76.10	1.23%	4.677
Transition Coverage	80.74	1.30%	4.556
Risk Reduction	62.63	1.01%	4.856
RR slightly aimed	63.84	1.03%	4.682
RR heavily aimed	62.90	1.01%	4.829
RR extremely aimed	61.26	0.99%	4.194

Table 5.6: Transition coverage. *This table shows that risk reduction has significantly worse results than random and transition coverage as the difference is more than two sigma. The transition coverage method is about one sigma better than random and is thus the best on transition coverage as expected.*

different traces with overlapping transitions the trace coverage does not show significant differences between risk reduction and transition coverage. Random testing however is significantly better than risk reduction in the trace coverage measure.

After investigating the cause of the differences in trace coverage we detected a similarity in the duplicates. All the duplicate traces contained only input actions. Most of the duplicates ended in a defect. The defects were detected because the model did not contain an output action that the IUT produced. Because the model did not have an output action the risk did not change after detecting the defect as only output action in the model have an influence on the risk values.

This seems to be caused by the way the algorithm for risk reduction works. It requires every test case to have at least one output action. If there is no output action the value of the reducible risk would stay identical. The identical risk values result in the risk reduction based method generating the same test case again and again until the IUT would return an output action. The random selecting between identically risked actions makes sure the method does not produce the same test case every time only input actions are performed.

We did expect other results for state coverage. Because a transition is defined as two states and an action we expected the state and transition coverage to have similar results, however the state coverage differences are hardly significant and are more like trace coverage. We do not

Method	# unique traces	Coverage	Sigma
Random	42.68	0.0159%	3.056
Transition Coverage	39.79	0.0148%	2.936
Risk Reduction	38.53	0.0143%	2.967
RR slightly aimed	38.84	0.0145%	2.267
RR heavily aimed	38.58	0.0144%	2.702
RR extremely aimed	38.32	0.0143%	2.562

Table 5.7: Trace coverage. *The results of the trace coverage are unexpected. transition coverage has about the same results as risk reduction. Random is however significantly better than risk reduction. These results are different than the results of transition coverage in table 5.6.*

know what causes this unexpected result.

Because we wanted to know the influence of the defects we also did a small side experiment on the same model but without defects. We expected these results to be identical to the model with defects because of the way we generate our test cases. If we would generate a test case before performing the test we would have seen a difference because different traces have the same path up to a defect but a different path after the defect. As a test stops when a defect is reached this would lead to multiple different traces being identical in practise.

We generate our test cases along the way because we cannot force an output action. Therefore we do not have different traces becoming identical due to defects. The coverage results should be identical for models with or without defects and this was confirmed by the results of our side experiment. The results were within half a sigma of the results on the model with defects.

CHAPTER 6

Conclusion

According to the results from the experiments, using the steering quality of the risk reducing method is a nice way of defining what parts of the system are important to test. Because of the way risk is defined the risk reducing method is capable of finding defects at a higher rate than other methods. It will also generate test cases that steer to output actions even if there are no important parts of the SUT defined.

We detected two possible problems with risk reduction based testing. The first being the lower coverage when compared against random or transition coverage methods. This is caused by the steering to output actions. If we steer to an output action but do not see an output action there is a very high chance we will perform the same trace multiple times. The other problem is the time risk reduction based testing requires for preparing the test cases. It takes too much compared with random or coverage based tests as they have no significant preparation time.

Using the results of the experiment we can now answer the research questions, beginning with the sub research questions.

6.1 Sub Research Questions

Sub research question one (SQ1) questions the ability to define a good initial risk. The experiment indicated that the results for detecting defects and reducting risk are good even if there are no specific parts defined as being important, e.g. using a uniform risk definition. Defining important parts does however make the framework run tests containing these important parts thus making it useful to steer.

The second sub research question (SQ2) asked to investigate the influence of risk based test set generation. When pointing to defects the method was able to reach these results. However the needed difference in initial risk for the aiming values has to be researched. Our results indicate a difference of at least 5 times the initial risk will produce a good steering mechanism. This was seen in the experiment where all our points of interest were reached. Because we were able to detect all types of defects the risk reduction based tests are not influenced by the type of defect. Defects do however prevent the test from continuing after the defect and thus result in a part of the IUT that cannot be tested. If the risk is high the risk reduction based method will have many duplicates in this area.

The last sub research question (SQ3) concerting the amount of overhead can be answered very simply: there is a lot of overhead. The overhead can be a problem for risk reduced testing. Risk reduced testing needs for a thorough preparation which takes time. According to our measurements performed during the experiment these preparations would take many years when performing the tests on a test set with a more usable test depth of 100 actions. This could be caused by a slow implementation or a defect in our implementation, however we are quite sure that even with a faster implementation the preparation time would be too much due to the high amount of different traces.

6.2 Main Research Question

Does the risk reduction based test give a better result than random or transition coverage when the tests are constrained by time or number of tests? Based on the results from QA1 we can say risk reduction is better at finding defects than random testing or transition coverage because there is a significant difference. However the results of QA4 indicate the coverage for risk reduction is significantly lower than the other methods. Because the coverage is low we have little use for the high amount of defects we can find with every unique trace. When we would combine QA1 and QA4 the result would be that the amount of defects we can detect is not significantly different.

When looking at the main qualities of risk reduction, testing important things (QA2) and reducing risk (QA3) we definitely see that risk reduction is significantly better at these qualities. But if we want to use risk reduction based testing because of these two qualities we note that the amount of preparation time required for the tests currently outweighs the benefits.

6.3 Future Work

Future research should target the significantly worse results for coverage to make use of the nice results for finding defects with risk reduction based tests. The coverage of risk reduction is low because finding a defect did not change the risks for the trace resulting in the same trace being executed multiple times. The algorithm does not handle the detection of defects. This can be handled by changing the implementation to leave out defective traces once they have been executed or to add the defective traces to the algorithm calculations as a zero risk trace, effectively removing them from the choices.

We also need to investigate the way we generate test cases. We cannot execute a test case that contains output actions because we can only force the IUT to perform input actions. This should be taken into account with the algorithm if we want to generate a test set. Generating a test set of more than one test case would increase the reduced risk and help get a higher coverage because the changing risks have no influence on the prepared test set.

If the algorithm has been corrected and we want to use it in practise we would also need to speed up the algorithm or its use. There are different ways of improving the algorithm or the way we should use it in practise.

The first method is not optimising the algorithm but changing the way it is used. We base this on the way chess computers calculate the next few possible moves and select the move with the best outcome [Mar87]. The same is usable for risk reduction based tests. When asked to make a choice we can calculate the next x levels of risk and select the action that could give us the best risk.

Another way of improving the calculations is to rewrite the algorithm to work in parallel. This does not lower the time per trace, but it does reduce the overall time. For example taking four general purpose graphical processing units (GPGPU) each capable of running 256 parallel processes would reduce the time required to do calculations with about 1000, in practise this results in 3-5 extra levels of depth that can be calculated.

And our last idea for an optimisation is to reduce the actual amount of calculations. A way of doing this is by using alpha-beta pruning [KM75]. Because the algorithm using a maximum selector to branch through the tree we need to see if there is a way of predicting which action will return the maximum value so we do not have to calculate the values for the other actions.

One way is by looking at previous calculated states. If a state has already been reached in a previous trace the maximum value will be identical because the following actions are identical. The risk value is however different due to the possibly different depth at which we reached the state.

Another way is by not calculating but just looking at the amount of output actions and their chance values. This could probably be a lot cheaper than calculating everything. We think this has potential and thus needs investigating.

Bibliography

[BBS06] Laura Brandán Briones, Ed Brinksma, and Mariëlle Stoelinga. A semantic framework for test coverage. In Susanne Graf and Wenhui Zhang, editors, *ATVA*, volume 4218 of *Lecture Notes in Computer Science*, pages 399–414. Springer, 2006.

[DJK+99] Siddhartha R. Dalal, Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton, Christopher M. Lott, Gardner C. Patton, and Bruce M. Horowitz. Model-based testing in practice. In *ICSE*, pages 285–294, 1999.

[DM90] Siddhartha R. Dalal and Colin L. Mallows. Some graphical aids for deciding when to stop testing software. *IEEE Journal on Selected Areas in Communications*, 8(2):169–175, 1990.

[DN84] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Trans. Software Eng.*, 10(4):438–444, 1984.

[KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[Mar87] T.A. Marsland. Computer chess methods. *Encyclopedia of Artificial Intelligence*, 1987.

[McC04] S. McConnell. *Code complete*. Microsoft Press Redmond, WA, USA, 2004.

[ST09] Mariëlle Stoelinga and Mark Timmer. Interpreting a successful testing process: Risk and actual coverage. In Wei-Ngan Chin and Shengchao Qin, editors, *TASE*, pages 251–258. IEEE Computer Society, 2009.

[Tre08] J. Tretmans. Model based testing with labelled transition systems. *Formal Methods and Testing*, pages 1–38, 2008.

[VDBRT04] M. Van Der Bijl, A. Rensink, and J. Tretmans. Compositional testing with ioco. *Formal Approaches to Software Testing*, pages 1102–1102, 2004.

[ZHM97] H. Zhu, P.A.V. Hall, and J.H.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):427, 1997.

APPENDIX A

Implementation

```
module RiskCoverage
  # Implementation of algorithm 1 from Stoelinga and Timmer's paper
  # Interpreting a Successful Testing Process: Risk and Actual
  # Coverage.
  #
  # The main function to call is the calculate_risk_based function.
  # It will run all the calculations for a depth of max_depth.


  ROOT = 1 # The state # for the root node, This is needed in the
           # code because the root node has a different data
           # structure than the other nodes

  # calculate the MT values up to a certain depth
  # @param max_depth, the depth up to which the risks are calculated
  def calculate_risk_based(max_depth)
    # Call the calculation function with the risk reduction depth
    # one higher than the state space depth. This because the lowest
    # depth returns risk reduction values of zero and thus making
    # them unusuable if we need to select the highest risk at depth
    # max_depth.
    calculate_rr_values([0],0,max_depth+1, Time.new.to_i)
  end

  # Loop over all the possible traces and calculate the risk
  # reduction value for all of them.
  # @param sigma, A list containing the current trace to calculate
  # @param depth, The current depth being calculated (sigma length-1)
  # @param max_depth, The depth at which to stop calulcating
  # @param time_id, An id to help keep track of different experiments
  def calculate_rr_values(sigma, depth, max_depth, time_id)
      current_node = node(sigma)

      # Depth first traversal, first handle the deepest nodes because
      # the nodes are dependant on there children.
      children = current_node.children

      if(depth < max_depth) and children.length > 0
        children.each {|child|
          new_sigma = sigma.dup
```

```
        new_sigma << child
        rr_value = calculate_rr_values(
          new_sigma, new_sigma.length - 1, max_depth, time_id)
        @mt_table[new_sigma.join("-")] = rr_value
      }
    end

    get_mt(sigma, max_depth - depth)
end

# Calculate the risk reduction value for an input or output action
# and take the maximum amount of risk reduction possible.
# @param sigma, The trace to calculate the risk reduction for
# @param depth, The current depth
def get_mt(sigma, depth)
  return 0.0 if depth == 0
  inp = doInput(sigma,depth)
  obs = observe(sigma,depth)
  [inp, obs].max
end

# Calculate the highest risk we can reduce using an input action
# @param sigma, The trace to calculate the risk reduction for
# @param depth, The current depth
def doInput(sigma, depth)
  return 0 if depth == 1

  # Get all the possible input traces
  current_node = node(sigma)
  choices = current_node.children.select{|child|
    child != ROOT and
    current_node.label_to_child(child).iotype == :input
  }

  return 0.0 if choices.empty?
  # Get the trace with the highest risk reduction value
  # This value is retreived from the cache.
  sigma_string = sigma.join("-")
  mts = choices.map {|choice|
    @mt_table["#{sigma_string}-#{choice}"]
  }
  mts.max
end

# Calculate the highest risk we can reduce when receiving an
# output action
# @param sigma, The trace to calculate the risk reduction for
# @param depth, The current depth
def observe(sigma, depth)
  # Get all possible output traces
  current_node = node(sigma)
  choices = current_node.children.select{|child|
    child != ROOT and
    current_node.label_to_child(child).iotype == :output
  }
```

```
  # risk reduction value at depth==0 is zero, thus no need to
  # calculate the mts value. They are the children of nodes at
  # depth==1. Also see the function calculate_risk_based(max_depth)
  if (choices.length==0 or depth==1)
    return get_r(sigma, get_obs(sigma))
  end

  # Because we cannot force an output, sum their risks together.
  sigma_string = sigma.join("-")
  mts = choices.map { |choice|
    @mt_table["#{sigma_string}-#{choice}"]
  }

  sum = mts.inject{|sum2, i| sum2 + i}
  get_r(sigma, get_obs(sigma)) + sum.to_f
end

# Count the number of times we have seen an output after sigma
# @param sigma, The trace to get all the outputs for
def get_obs(sigma)
  current_node = node(sigma)
  children = current_node.children.select{|child|
    child != ROOT and
    current_node.label_to_child(child).iotype==:output
  }

  obs = 0
  children.each {|child|
    obs += current_node.times_to_child(child)
  }
  obs
end

# Calculate the risk added when observing after a trace
# @param sigma, The trace after which we will observe
# @param m, the amount of previous observations
def get_r(sigma, m)
  total1 = get_c(sigma, m)
  total2 = get_c(sigma, m + 1.0)
  total1 - total2
end

# Calculate the risk contribution of a certain sigma with a given
# previous observing rate.
# @param sigma, The trace after which we will observe
# @param m, An observation rate
def get_c(sigma, m)
  preach = get_preach(sigma)

  sum = 0.0
  (0..m).each{|i|
    mult = combination(m,i) * (preach**i)
    mult *= ((1.0 - preach)**(m-i))
    mult *= get_PEP(sigma, i)
    sum += mult
  }
```

```
    new_sigma = sigma.dup
    get_w(new_sigma) * sum
  end

  # Get the fault severity value for a certain trace sigma.
  # W makes sure the depth is taken into account for in the risk
  # For the experiments it is simplified as 100 * 0.05^(depth-1)
  # @param sigma, The trace to calculate the W value for
  def get_w(sigma)
    # The root has a fault weight of 100
    return 100 if sigma.length < 2

    # labda is a value between 0 and 1/m
    # m = maximum outdegree of sigma
    # This assures the total risk is finite.
    labda = 0.05
    sigma.pop
    get_w(sigma) * labda
  end

  # Calculate the Posterior Error Propability
  # PEP is the chance a trace (sigma) has been implemented wrong
  # even though the test with the trace passes as being correct.
  # @param sigma, the trace to calculate the PEP for
  # @param obs, the observation count.
  def get_PEP(sigma, obs)
    perr = get_perr(sigma)
    pfail = get_pfail(sigma)

    # **obs is needed to make sure the risk takes into account the
    # previous passes of an output action.
    pepw = ((1.0-pfail)**obs) * perr
    pepw /= (((1.0-pfail)**obs) * perr + 1.0 - perr)
    pepw
  end

  # The chance a trace is implemented wrong.
  # The values are stored in a lookup table.
  # @param sigma, The trace to get the chance for
  def get_perr(sigma)
    from = sigma[sigma.length-2]
    to = sigma[sigma.length-1]
    @statespace[from].get_perr(to)
  end

  # The chance a defective trace is actually noticed.
  # The values are stored in a lookup table.
  # @param sigma, The trace to get the chance for
  def get_pfail(sigma)
    from = sigma[sigma.length-2]
    to = sigma[sigma.length-1]
    @statespace[from].get_pfail(to)
  end
```

```
  # Get the chance that the given action is seen.Input actions have
  # a chance of 1 and output actions are specially calculated
  # @param sigma, The trace to the current state
  # @param next_child, The target state representing the action
  def get_pout(sigma, next_child)
    current_node = node(sigma)

    if(current_node.label_to_child(next_child).iotype==:input)
      return 1.0
    end

    if current_node.chance_to_child(next_child).nil?
      calculate_chance_for_output(sigma[sigma.length-1], next_child)
    end

    current_node.chance_to_child(next_child)
  end

  # Determine the chance to receive a certain output action.
  # This function can be used to give a more realistic
  # represenatation of the model and its output actions.
  # @param from, Current state
  # @param to, The state we will be in after an output action
  def calculate_chance_for_output(from, to)
      from_node = @statespace[from]
      output_children = from_node.children.reject{|child|
        from_node.label_to_child(child).iotype==:input
      }
      chance = 1.0/output_children.length
      from_node.set_chance_to_child(to, chance)
  end

  # Preach is the chance we will can execute/expect a trace
  # @param sigma, The trace to be reached
  def get_preach(sigma)
    return 1.0 if (sigma.length<2) # It is the root!

    total = 1
    (2..sigma.length-2).each {|i|
      total * get_pout(sigma[0..i], sigma[i+1])
    }
    total

  end

  # Return the node belonging to the last item of sigma.
  # @param sigma, Trace to get the last node from.
  def node(sigma)
    node_id = sigma[sigma.length-1]
    @statespace[node_id]
  end
end
```